# Automatic Pool Allocation for Disjoint Data Structures *

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu

**Abstract —** This paper presents an analysis technique and a novel program transformation that can enable powerful optimizations for entire linked data structures. The fully automatic transformation converts ordinary programs to use pool (aka region) allocation for heap-based data structures. The transformation relies on an efficient link-time interprocedural analysis to identify disjoint data structures in the program, to check whether these data structures are accessed in a type-safe manner, and to construct a Disjoint Data Structure Graph that describes the connectivity pattern within such structures. We present preliminary experimental results showing that the data structure analysis and pool allocation are effective for a set of pointer intensive programs in the Olden benchmark suite. To illustrate the optimizations that can be enabled by these techniques, we describe a novel pointer compression transformation and briefly discuss several other optimization possibilities for linked data structures.

## 1. INTRODUCTION

Pointer-intensive programs are a difficult challenge for modern processor architectures and memory hierarchies. Typical pointer-based heap data structures produce access patterns with much worse locality than dense arrays because they are not accessed in a regular pattern within the linear system address space. Furthermore, the addresses of future accesses are difficult to predict, making prefetching difficult to apply. Traditional optimizations have generally focused on optimizing individual data objects and access patterns [6, 27, 19, 4], but have generally neglected to take a more macroscopic approach that analyzes and transforms entire logical data structures. High level optimizations have generally been limited to runtime techniques such as clustering, profile-driven layout optimization, and cache-conscious garbage collectors [9, 31, 7, 23, 6]. We have developed a macroscopic approach to optimizing linked data structures that can enable sophisticated compile-time transformations of entire data structures, even for unsafe languages like C.

*This work is sponsored by an NSF CAREER award, grant number EIA-0093426, and supported in part by the NSF Operating Systems and Compilers program under grant number CCR-9988482 and by an equipment donation from Hewlett Packard.

This paper describes a technique for introducing *fully automatic* pool allocation of heap-allocated data structures in general C programs. Pool allocation[1] is often used manually by programmers to increase program performance because pool allocators are often more efficient than general purpose allocators, and the resulting allocation patterns often have better memory locality. Fully automatic pool allocation provides the same benefits, but also provides the compiler a basis for performing new transformations that optimize entire logical data structures, as discussed below.

Although pool allocation is a commonly applied manual optimization, we are not aware of any previous compiler work that automatically introduces pool allocation into general programs (e.g., C programs using malloc and free). Existing techniques for automatic pool allocation are primarily runtime techniques that use heuristics to segregate objects into pools by size, type, or predicted lifetimes [15, 23]. The closest example to our work [2] uses profiling to identify allocations with short lifetimes and then places these in fixed size regions. These techniques generally do not take into consideration the connectivity of allocated nodes, and do not provide any basis for new compile-time transformations (Language support for manual region-based allocation has also been proposed [13] and could be used for macroscopic compile-time transformations, but requires manually inserted annotations to direct it).

The key to the automatic pool allocation transformation is identifying logically disconnected data structures and their interconnection properties, so that we can assign disjoint data structures to different memory pools. We use an analysis described in Section 2 to identify logically disjoint data structures, and use it to compute a representation we call the Disjoint Data Structure Graph. Our analysis to construct these graphs is similar to previous work on heap connection analysis and shape analysis [18, 17, 14, 30, 22], but differs from that work in a few key ways (discussed in more detail in Section 6). We perform the data structure analysis and subsequent transformations entirely *at link-time*, using a compilation framework called LLVM, described briefly in subsection 1.1. Link-time is an appropriate place for our data structure analysis, because it is fundamentally interprocedural.

Once the Disjoint Data Structure Graphs have been computed, the program is transformed to use pool allocation, as described in Section 3. This transformation can only be applied to logical data structures that are accessed in a type-safe manner. If the graph for the data structure contains multiple different node types, a separate pool is used for each node type so that each pool contains only homogeneous objects (Although this is not necessary, it can significantly speed up allocation and deallocation of memory from each pool). The runtime system uses pool descriptors to record rele-

[1] Sometimes called region-based allocation [13].

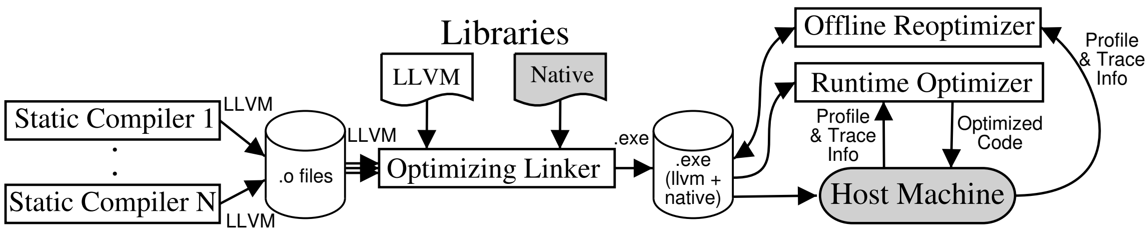



**Figure 1: LLVM system architecture and compilation process**

vant bookkeeping information, including the connectivity between linked pools. The compiler inserts code to allocate and destroy the pool descriptors for each logical data structure instance within the "root" function that entirely contains the lifetime of the data structure. The compiler then rewrites all the allocation and deletion operations for that data structure to allocate and free heap objects out of the appropriate pools. The entire memory of a pool as well as the pool descriptors are released back to the system at the point that the data structure it contains is no longer accessible (i.e., at the exit from the root function). Section 4 presents experimental results on the effectiveness of the data structure analysis and pool allocation transformations.

The disjoint data structure analysis combined with automatic pool allocation can enable sophisticated optimizations for linked data structures, because the compiler has much more information about allocation patterns and locality properties of nodes allocated from the pools. We briefly describe a few such optimizations in Section 5, including a novel *automatic pointer compression* transformation, however evaluating these transformations is beyond the scope of this paper. The pointer compression technique addresses a key problem in 64-bit architectures — pointer-intensive data structures pay a significant penalty in terms of memory size (and therefore, cache efficiency and memory bandwidth) because of using 64-bit pointers. The transformation we propose *transparently* and *safely* replaces 64-bit pointers within each logical pool-allocated structure with smaller (16 or 32-bit) offsets into the pool, and dynamically changes pointer sizes and rewrites the pool contents as the data structure grows. One of the goals of our future work is to implement and evaluate the pointer-size compression and other optimizations enabled by automatic pool allocation.

## 1.1 The LLVM Compilation System

Our work has been implemented within the LLVM compilation system. LLVM — Low Level Virtual Machine — is a compilation system designed to support high level optimizations for *link-time*, *runtime*, and *offline* compilation. The key idea in LLVM is to use a rich virtual instruction set (instead of raw machine code) as the code representation manipulated by a link-time optimizer and code generator. The LLVM instruction set uses low-level RISC-like operations, but provides rich information about the operands, including extensive language-independent type information and dataflow information in Static Single Assignment form. The LLVM compilation strategy is depicted in Figure 1, and an example C function and it's corresponding LLVM assembly code are shown in Figure 2.

The LLVM instruction set and types are low-level enough to represent programs from any source language, enabling arbitrary source-level compilers to generate LLVM object code instead of machine code. Once linked, the LLVM code provides important information about the program that is not present in machine code, such as type and dataflow information. This is important because interprocedural optimization is much more convenient at link-time than during source-level compilation (since the latter can require significant changes in the development process to make complete or nearly complete application source code available). Traditional compilation strategies (which only manipulate low-level machine code at link time) make it difficult to perform high-level analyses and transformations due to lack of high-level information[2] [3, 1].

We have written an LLVM backend for GCC, which currently allows compilation of C to LLVM and will support C++ in the near future. The LLVM infrastructure includes standard scalar optimization passes and an optimizing linker. The optimizing linker links the different LLVM object code files (along with any libraries that have been compiled into LLVM object code), performs interprocedural optimizations on the resulting program, and then generates native machine code for a SPARC v9 architecture. We are currently developing the runtime and offline optimizers shown in the diagram. The analysis and transformations described in this paper are implemented in the LLVM optimizing linker.

# 2. DATA STRUCTURE ANALYSIS

Automatic pool allocation requires a program analysis which exposes the allocation pattern of the program, the connectivity graph of allocated memory objects, and indicates whether or not the transformation is provably *safe* to perform. Additionally, such analysis (which is fundamentally interprocedural) must be efficiently computable if automatic pool allocation is to be feasible in practice. The data structure analysis graph is our representation for solving these problems.

Our implementation of the data structure analysis algorithm is implemented in the context of the LLVM system, but should be easily adaptable to other systems with similar analysis capabilities. The data structure analysis should also be directly applicable to static or link-time compilers for Java bytecode or the Microsoft Common Language Runtime (CLR), which have rich, high-level bytecode representations from which the necessary analysis information can be extracted.

The key analysis information we use is as follows:

- SSA form – We assume a low-level code representation with an infinite set of virtual registers, and a load-store architecture (memory locations can be accessed only via load and store operations). The virtual registers are assumed to be in SSA form, but the memory locations are not. We assume that

[2] In fact, some commercial compilers export the static compiler's internal representation in order to enable more sophisticated link-time optimizations [1, 8, 12]. In contrast, LLVM provides a simpler, and more elegant solution that can work with arbitrary source-level compilers, enabling sophisticated high-level optimizations.

```
/* C Source Code */
struct Patient { ... };
typedef struct List {
  struct List    *forward;
  struct Patient *patient;
  struct List    *back;
} List;

void addList(List *list, struct Patient *pt) {
  List *b = NULL;
  while (list != NULL) {
    b = list;
    list = list->forward;
  }
  list = (List *)malloc(sizeof(List));
  list->patient = pt;
  list->forward = NULL;
  list->back    = b;
  b->forward    = list;
}
```

```
;; LLVM assembly code
%Patient = type { ... }
%List    = type { %List*, %Patient*, %List* }

void %addList(%List* %list, %Patient* %pt) {
bb0:
  %cond1 = seteq %List* %list, null
  br bool %cond1, label %bb3, label %bb2
bb2:
  %list1 = phi   %List* [%list2,%bb2], [%list,%bb0]
  %list2 = load  %List* %list1, uint 0
  %cond2 = setne %List* %list2, null
  br bool %cond2, label %bb2, label %bb3
bb3:
  %b     = phi   %List* [%list1,%bb2], [null,%bb0]
  %list3 = malloc %List
  store %Patient* %pt,    %List* %list3, uint 1
  store %List*    null,   %List* %list3, uint 0
  store %List*    %b,     %List* %list3, uint 2
  store %List*    %list3, %List* %b,     uint 0
  ret void
}
```

**Figure 2:** **A C function and the corresponding LLVM assembly code**

it is impossible to take the address of an SSA virtual register. Using SSA form is not essential, but it simplifies the analysis because register values cannot be killed, and it allows for efficient identification of uses of values.

- Identification of memory objects – We assume that the compiler can distinguish four types of memory objects, namely, heap objects allocated by `malloc`, stack objects allocated by `alloca`, global variables, and functions; and can identify the declared types of each object.

- Type information – We assume that all SSA variables and memory objects have an associated type. The type system only needs to distinguish primitive types (e.g., integer and floating point), pointers, structures (i.e., user-defined aggregates), and arrays. Only memory objects can have structure and array types, i.e., only `load`, `store`, `malloc` and `alloca` operations are possible on such types. All SSA variables must be of primitive or pointer type.

- Safety information – Our analysis requires that there is some way to distinguish type-safe and type-unsafe usage of data values. In LLVM, all operations (including memory load/store operations) follow strict type rules, e.g., it is illegal to add an integer and a floating point number, or to perform arithmetic on a pointer type. LLVM includes a `cast` instruction that can be used to convert between types (e.g., before adding an integer and a float), but most such cast operations do not actually violate type safety for the underlying data values. A specific form of this instruction (casting to a pointer type) is the *single* way that type-unsafe operations can be performed in LLVM. This mechanism is sufficient to implement arbitrary unsafe code, and makes violations of type safety trivial to detect.

## 2.1 The Disjoint Data Structure Graph

The Disjoint Data Structure Graph (or simply data structure graph) is the primary representation of the access patterns of a function, library, or whole program. Each node in the graph represents a typed SSA register or a memory object allocated by the program, or multiple objects of the same type. A node is represented by a node type (described below), the program type of the memory object, and a set of fields. A node contains one field for every primitive or pointer value contained in the object, including fields of nested structures. An array, however, is represented as a node with a single field, i.e., our analysis does not track individual array elements. Note that an ordinary pointer variable is represented in the graph as a node with a single field, for uniformity.

Each edge in the graph connects a pointer field of one node (the source field) to a field of another node (the target field). The souce or target of an edge is represented as a <node, field-index> pair. A pointer field may have edges to multiple targets, i.e., edges represent "may-point-to" information. In practice, however, a pointer field usually has a single outgoing edge because of the node-merging technique described in Section 2.2.3.

The finished analysis graph for the `addlist` function (defined in Figure 2) is shown in Figure 3. Its construction will be used as the running example to explain the data structure graph.

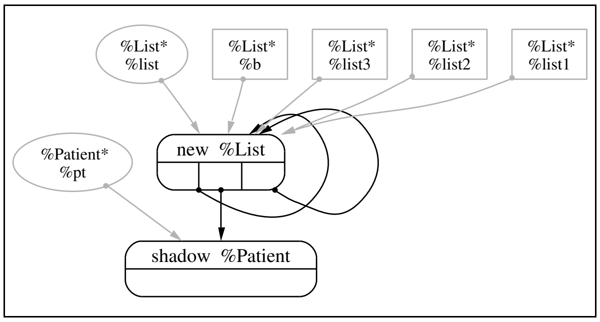



**Figure 3:** **Data structure graph for** `addList`

This data structure graph shows the graphical notation we will be using to illustrate the analysis results. In our graphs, the dark rounded objects represent actual memory objects that exist in the program, whereas the lighter objects represent scalar values in the function (elliptical scalar values indicate incoming arguments). The three pointer fields of the `%List` type are reflected directly in the data structure graph as three fields in the `%List` memory object.

We use eight distinct node types to distinguish different kinds of objects:

1. `new` node – Represents memory allocated on the heap with the `malloc` operation.
2. `alloca` node – Represents memory allocated on the stack of the current function (either by the `alloca` instruction or because the address of an automatic variable must be available).
3. `global` node – Represents the memory occupied by a global variable.
4. `function` node – Represents a function in memory. These nodes serve as the targets of function pointers.
5. `call` node – Represents a call to a function. Contains a field holding one or more pointers to the called `function` node(s), plus fields holding pointers to the formal arguments.
6. `shadow` node – Represents memory that we know exists, but don't know how it is allocated.
7. `cast` node – Represents memory used in a non-typesafe manner.
8. `scalar` node – Represent SSA registers of pointer type in the current function. These nodes have a single field that holds the set of memory objects a scalar pointer may point to. Because we assume that it is impossible to take the address of an SSA register, there may be no edges pointing to scalar nodes.

For the purposes of analysis, the `alloca` and `new` nodes are treated identically, and referred to as "allocation" nodes. The `global` and `function` nodes are also treated identically, since all functions bodies are global objects (with no pointer fields).

`Shadow` nodes are special nodes that are used to represent cases where memory is allocated outside of the scope of the analysis, but referenced in the current function. Because we don't know what kind of memory it is (global variable or an allocation), we cannot assign it a concrete node, so we make it a special `shadow` node.

The data structure graph is computed directly from the LLVM source. The key goal of the analysis is to construct a data structure graph for each function in the program, identifying distinct logical data structures that are visible to each function. For pool allocation and other transformations, it is important to separate *disjoint* data structures into distinct logical data structures instead of conservatively merging them. For this reason, our analysis is context-sensitive.

The strengths of our analysis include identification of disjoint data structures, small analysis time, concise summarization of important information, unified handling of both heap and stack allocations, and a clear splitting of the analysis between intra- and inter-procedural analyses. Splitting the analysis into an two different stages allows the intraprocedural portion of the analysis to be performed separately for each module, requiring only the interprocedural portion to be executed at link-time. Section 2.2 describes the intraprocedural analysis, used to build the initial version of the data structure graph for each procedure. Section 2.3 describes the interprocedural analysis used to compute the interprocedural closure of the graph.

## 2.2 Intraprocedural Analysis Algorithm

The intraprocedural graph computation phase is a *flow-insensitive* analysis that builds a data structure graph without requiring the code for other functions to be available. The result of this algorithm is the data structure graph, and the returned pointer set. The returned pointer set is used to track which pointers values may be returned from the function, and is used when computing the interprocedural closure of the graph.

The graph construction algorithm is composed of three distinct phases: the node discovery phase, the worklist processing phase, and the graph simplification phase. These analyses are performed directly on the LLVM code representation, making extensive use of the type information provided. Although our analysis is flow insensitive, the use of SSA form inherently provides a degree of flow sensitivity and reduces the occurrence of spurious edges in the graph.

### *2.2.1 Node Discovery Phase*

The node discovery phase performs a single pass over the function being processed, creating the nodes that make up the graph. Specifically, it generates a node of the appropriate type for each function call, allocation (stack or heap), and global variable referenced. It creates `cast` nodes to represent unsafe pointer conversions and scalar nodes for SSA pointer registers. Shadow nodes are created for values that point to unknown values (incoming pointer values and the return value of function calls that return pointers). The worklist processing phase can only add new shadow nodes and edges to the graph, so all nodes of other types come from the node discovery phase.

As nodes are generated, all SSA variables that point to the new nodes (for example, the `%list3` variable) have their outgoing edges updated, and all *uses* of these SSA variables are put on the worklist. The graph computed for `addList` after the node discovery phase is shown in Figure 4. Note that none of the fields of the allocated nodes have outgoing edges yet, nor do the scalar values `%list1`, `%list2`, and `%b`.

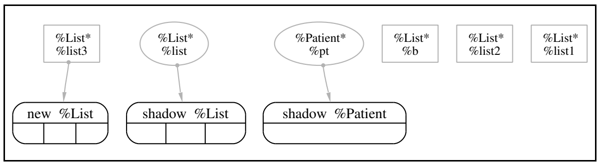



**Figure 4:** `addList` **graph after node discovery**

The worklist currently contains `%cond1` and `%list1` (which use %list), and the four store instructions (which use `%list3` and `%pt`).

### *2.2.2 Worklist processing*

On entry to the worklist processing phase of the algorithm, the node discovery phase has created data structure nodes for all of the non-shadow nodes (and some shadow nodes) in the function graph. The worklist contains all of the instructions in the function that use the SSA values corresponding to the nodes. Worklist processing consists of popping an instruction off the worklist, adding edges to the data structure graph, and putting instructions onto the worklist if they refer to fields that have been updated (the operations needed when updating a single scalar variable or structure field are captured in the UpdateScalar and UpdateField functions described below):

**ProcessWorkList**()
  while $WL \neq \emptyset$
    instruction $inst$ = $WL.head$; $WL$.remove($inst$)
    *process instruction*($inst$)

We describe the effects of each primitive that may be found on the worklist, giving the C syntax for the operation, short pseudocode for the update, and a description. In addition to the instructions below, there may also be `free` calls and pointer comparison operations on the worklist, but since they do not affect the data structure graph they are ignored. The other instructions are processed as follows:

- `X = &A[i]` appears below; first:
- X = &(*Y).field<N>
  UpdateScalar($X$, Advance(PointingTo($Y$), $N$))

  Taking the address of a structure field is implemented by simply advancing the source pointers ($Y$) to point to the new field. Since pointers are represented as <node, field-index> pairs, the $Advance$ function simply increments the $field\text{-}index$ portion of each pointer in the input set.

- `X = &A[i]`
  UpdateScalar($X$, PointingTo($A$))

  Our analysis ignores array subscripts, effectively treating entire arrays as a single element. This is conservative, but allows for a compact representation with few implementation difficulties.

- `*X = P`
  if IsPointerType(P) then
  UpdateFields(PointingTo($X$), PointingTo($P$))

  Storing a pointer into a memory object adds the edges emanating from the node for $P$ to the fields pointed to by $X$.

- `X = *P`
  if IsPointerType(X) then
  if ContentsOf(PointingTo($P$)) $= \emptyset$ then
  $SN$ = new ShadowNode(Type($*X$))
  UpdateFields(PointingTo($P$), $SN$)
  UpdateScalar($X$, ContentsOf(PointingTo($P$)))

  Loading a pointer from memory updates the scalar to also include the contents of the field in memory. If there is no outgoing edge from one of the fields we are loading from, however, we must *synthesize* a shadow node and add an edge from the field to the new shadow node. This ensures that the load instruction will have a node to point to, but it is also important to make sure that edges are not lost in the graph. The ramifications of this technique are discussed in the example, below.

- X = $\phi$(Y, Z)
  UpdateScalar($X$, PointingTo($Y$) $\bigcup$ PointingTo($Z$))

  The SSA $\phi$ instruction merges together values due to control flow. We simply union together the pointer sets.

- `call func(P)`
  UpdateParam($callnode$, $argnum$, PointingTo($P$))

  Processing a function call simply updates the call node to keep track of the accurate pointer sets that are being passed in as parameters. This is used for the interprocedural closure stage, described in Section 2.3. This does not cause any worklist entries to be modified.

- `return X`
  $ReturnPtr = ReturnPtr \bigcup$ PointingTo($X$)

  This adds the nodes pointed to by $X$ to the return set for the function.

The "PointingTo" function returns the set of pointer values stored in the scalar node for the specified variable. The "UpdateScalar" function updates the scalar node corresponding to the specified variable to include all of the pointers in the set specified as its second argument. If this causes the scalar node to change, all of the uses of the SSA scalar are added to the worklist for reprocessing, and the graph simplification algorithm is run:

**UpdateScalar**(scalar $S$, pointerset $PS$)
  if edges($S$) $\bigcap PS \neq PS$ then
    $WL$.add(uses($S$))   // Add all uses to worklist
    edges($S$).add($PS$)   // Add new edges to graph
    GraphSimplify($S$)   // Merge nodes (Section 2.2.3)

The "UpdateFields" function is identical, but updates the fields of memory objects pointed to by the first argument. If the fields change, all loads refering to the structures are added to the worklist, and the graph simplification algorithm is run.

In our example, after processing the `%cond1` instruction (which is a no-op) and `%list1` instructions, the `%list2` instruction is added to the worklist, and an edge from `%list1` to the shadow `%List` node is added. When processing the `%list2` instruction (a load), a shadow node must be synthesized, resulting in the graph shown in Figure 5 (additionally, the `%list1` instruction is reinserted into the worklist). Figure 5 shows the data structure graph after processing these instructions and `%list1` again.

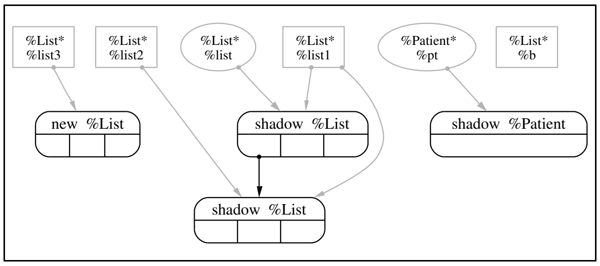



**Figure 5:** `addList` **graph after some worklist steps**

At this point, the worklist contains `%list2` and `%b` (which use %list1), and the four store instructions (which use `%list3` and `%pt`).

Clearly, the uncontrolled creation of shadow nodes can lead to infinite recursion: processing `%list2` again would cause another shadow node to be created (due to the load from the empty field on the shadow node created in the last iteration), which would cause future shadow nodes to be created – ad infinitum. To prevent this infinite recursion, and to ensure the data structure graph remains as compact as possible, the graph simplification algorithm is used to merge nodes in the graph that are indistinguishable from each other.

### 2.2.3 Graph Simplification Algorithm

In Figure 5, the two `%List` shadow nodes are *indistinguishable* from each other, and should be merged. Two nodes are considered indistinguishable if they are of the same LLVM type, if they have the same node type (or if at least one is a shadow node), and if there is a field in the data structure graph that points to both nodes. When two nodes are merged, the resulting node contains the union of the edges in the two original nodes, preventing edges from getting dropped from the graph. Additionally, any pointers to either original node get updated to point to the new merged node (this is similar to approaches taken by other analyses[14, 22, 24]).

The intuition behind this heuristic is if a pointer contains edges to two different nodes, all stores and loads that use the pointer will cause their effects (either edges added to the graph, or scalars updated with the field contents) to happen to both nodes at the same time. Since each node will eventually contain the same contents as the other, they might as well be merged.

In the case of `addList`, the `%list1` scalar points to both shadow nodes, and they are of compatible LLVM and node type. Merging the two nodes together causes the edges connecting the two nodes to become a self-loop, providing information about the connectivity. Important to note is that this summarization of the graph does lose information. For example, by merging the two nodes, we lose the information that `%list` only points to the first node. However, because we are dealing with potentially infinite structures without infinite time and space resources, we must perform some form of summarization.

In practice, we have found this information loss to be of little importance to our data structure analysis. This is suprising, because results on pointer analysis have shown Steensgaard's algorithm[24] as being significantly less precise than other approaches, and it uses a similar merging technique. The difference is that we are attempting to extract data structure connectivity information, *not* aliasing information. Pool allocation actually benefits from graphs that are merged as much as possible, as long as two disjoint structures are not unneccesarily merged together.

Figure 6 shows the data structure graph after merging the two nodes, processing `%b` and the four store instructions (which add outgoing edges to the "new" node as well as the remaining shadow node).

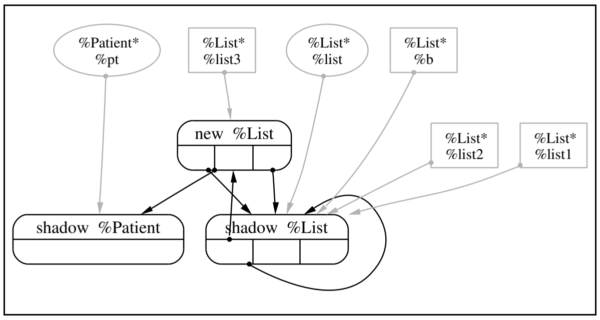



**Figure 6:** `addList` **graph after node merging**

Processing the last store in the function causes the edge to the "new" node to be added to the shadow node. Since there is now a field that points to two indistinguishable nodes (the new node and the shadow node), the two nodes are merged, yielding the finished graph shown in Figure 3. This merging behavior is an important part of our analysis — adding extra edges (for example, introduced by conservative heuristics) to the graphs cause them to *shrink* in size.

## 2.3 Interprocedural Closure Algorithm

The local analysis graph is of limited usefulness for program transformation, because most interesting data structures are passed to functions to construct or manipulate them (for example see `ProcessLists` in Figure 8). Without interprocedural information, it is impossible to transform these functions, because one of the called functions may use the data structure in ways not reflected in the local data structure graph. To indicate that function calls are involved, `call` nodes are introduced into the local data structure graph (shown in Figure 9). When the data structure graphs for the called functions are available, the call nodes are eliminated, being replaced with the analysis information in the called function graph itself. Thus, the interprocedural closure algorithm is the process of resolving `call` nodes to the graphs that they represent, propogating interprocedural information from the called functions to the caller function's graphs.

In general, each `call` node (and therefore, the `call` instruction it represents) may be able to call multiple functions if it is an indirect call. Our analysis handles this in a simple and uniform way. Functions are considered to be global objects themselves (and are therefore represented with a node in the data structure graph) and all `call` nodes have a field that indicates which functions are called by the `call` node. The common case is to call a single function directly, but there may be a call to a shadow function (if a function pointer is passed into the current function), or may contain edges to multiple functions (if its an indirect call).

**ComputeInterproceduralClosure**(graph $G$)
  $InlinedFnsSet = \emptyset$
  while $\exists$ node $Call \in$ CallNodes($G$)
    scalarnode $CallValue$ = ScalarFor($Call$)
    if $Call \in InlinedFnsSet$ then
      ResolveNodeTo($CallValue$, $InlinedFnsSet[Call]$)
    else   // Haven't already resolved this function
      $ReturnPtr = \emptyset$
      $\forall$ node $Fn \in$ TargetFunctions($Call$)
        $<retptr, argptrs>$ = Inline($Fn$, $G$)
        $ReturnPtr = ReturnPtr \bigcup retptr$
        // Resolve argument shadow nodes to call parameters
        $\forall$ ptrset $Arg \in argptrs$
          ResolveNodeTo($Arg$, Param($Call$, $Arg$))
      $InlinedFnsSet[Call] = ReturnPtr$
      // Resolve returned value to union of returned pointers
      ResolveNodeTo($CallValue$, $ReturnPtr$)
      RemoveFromGraph($Call$)
      RemoveUnreachableNodes($G$)

// "unknown" node $N$ is now points to $PtrVal$ set
**ResolveNodeTo**(scalarnode $N$, ptrset $PtrVal$)
  edges($N$).add($PtrVal$)
  GraphSimplify($N$)
  RemoveFromGraph($N$)

**Figure 7: Interprocedural Closure Algorithm**

The interprocedural closure algorithm (defined in Figure 7) operates by looping over the `call` nodes in the specified graph, inlining each called function's graph in place of the call node. As it does so, it uses the `ResolveNodeTo` function to resolve any argument shadow nodes to the actual parameters that are passed in. This analysis handles indirect call nodes by repeatedly inlining the called function graphs for each function called by a particular call node. If the called function returns a pointer, the return value set of the inlined function graph is used to eliminate the shadow node generated for the call node.

The core of the interprocedural closure algorithm is the "Inline" function. This graph inliner copies all of the nodes from the graph of the specified function into another graph, preserving edges between nodes. The `scalar` nand `alloca` nodes of the inlined function are discarded and their incident edges are removed from the graph (the values they represent are out of scope, so they cannot be referenced). The return value set and the set of argument nodes from the inlined graph are returned as a pair. The result of inlining a function call is memoized in the *InlinedFnsSet* to avoid infinite recursion when inlining recursive functions.

Although it is possible to simply inline the intraprocedural datastructure graph for a function, the inliner prefers to inline the interprocedural graph for the specified function if it is available. The only scenario where the graph is unavailable is when we are inlining a function call to a mutually recursive function, in this case, the intraprocedural graph must be selected to avoid infinite recursion. To get this effect, we calculate the interprocedural closure graphs in a postorder traversal over the call graph.

The "ResolveNodeTo" function merges a node with a set of nodes referenced by a pointer value, and is used to eliminate shadow nodes. This function takes a node argument and a set of pointer values to resolve. The node argument is either the `scalar` node that captures the return value of a call, or is a pointer argument in the inlined function that is now being resolved. The "ResolveNode" function operates by using the "GraphSimplify" algorithm to merge the node sets together, eliminating the shadow nodes inserted during the node discovery phase.

As a example to show the interprocedural closure algorithm, consider the `ProcessLists` function:

```
void ProcessLists(unsigned N) {
  List *L1 = malloc(sizeof(List));
  List *L2 = malloc(sizeof(List));
  unsigned i;
  /* populate lists */
  for (i = 0; i != N; ++i) {
    addList(L1, malloc(sizeof(Patient)));
    addList(L2, malloc(sizeof(Patient)));
  }
  useLists(L1, L2);   /* Use lists */
}
```

**Figure 8: Source for `ProcessLists` function**

In this example, there are three function calls. For purposes of discussion, assume that we have the data structure graph for `addList` before, but we do not have the graph for `useList` yet. In this case, the two `addList call` nodes in the intraprocedural data structure graph (Figure 9) may be resolved to actual graphs that they correspond to.

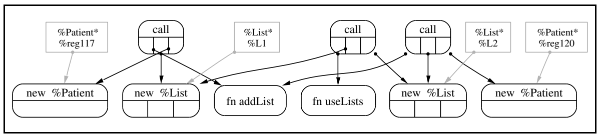



**Figure 9: Intraprocedural `ProcessLists` graph**

Inlining the function graph for the first `addList` call results in one of the Patient and one of the List nodes being used to resolve the incoming arguments of the `addList` function. After the graph is inlined, it is simplified (see Section *2.2.3* which merges the existing shadow Patient node with the Patient node allocated in this function. The second call is inlined in an identical manner, producing the finished graph shown in Figure 10.

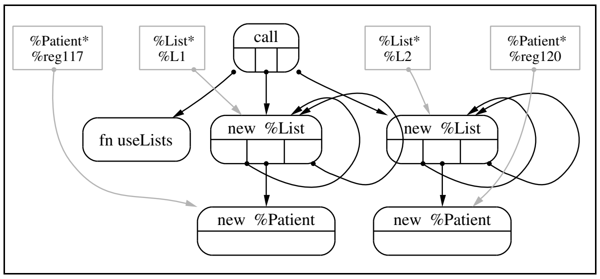



**Figure 10: `ProcessLists` graph with resolved calls**

This example shows three critical aspects of our algorithm that differentiates it from existing work. First, it is *incremental*, allowing programs to be analyzed as they are brought together. In the case above, perhaps the `addList` and `ProcessLists` function were in the same translation unit, allowing the graph to be partially resolved during compilation. Presumably, the `useLists` function graph will later be resolved at link time.

The second important property is that the algorithm preserves *disjoint* data structures. In the example above, the two disjoint linked lists are not conservatively merged together just because they are built with the same function (the `addList` function allocates all but one of the nodes in the list). This property is important for pool allocation because it means that we can more accurately allocate data structures into pools.

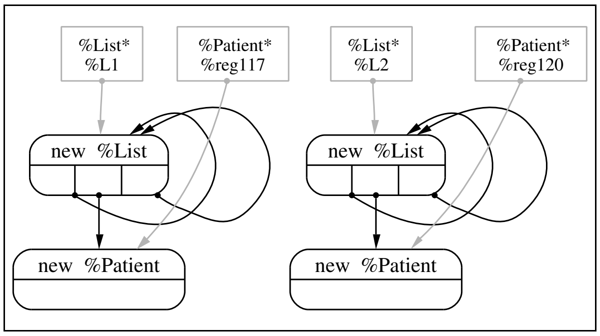



**Figure 11: The finished `ProcessLists` graph**

The third important property is that the data structure graph for a function is small, only including nodes that are reachable in the current function. For example, assume that the `useLists` function is an empty function, which would resolve to the graph shown in Figure 11. In this case, we know that the lifetime of the two lists are bounded by the lifetime of `ProcessLists` function itself, because they do not escape from the function. If we consider another function (say `main`) that calls the `ProcessLists` function, the two lists in Figure 10 will not be reachable in the `main` graph when the call to `ProcessLists` is resolved. For this reason, the nodes are eliminated, reflecting the fact that `main` is not exposed to the internal data structures used by `ProcessLists`.

# 3. AUTOMATIC POOL ALLOCATION

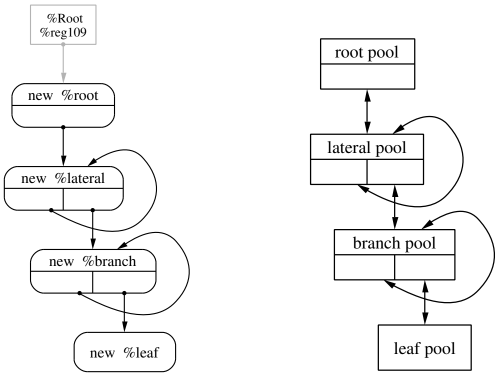
Many other researchers have illustrated the value of pool allocating data structures [10, 13, 26], but *fully automatic* transformation is a challenging problem. Here we describe a simple algorithm for automatic pool allocation of C programs that uses the data structure graph to ensure safety of transformation.

## 3.1 Runtime support

We designed a simple pool allocation runtime library with four external functions (`poolinit`, `pooldestroy`, `poolalloc`, `poolfree`), and one data type (the pool descriptor). We transform the program to pass the pool descriptors into functions that must allocate or free nodes from or to the pool. In this way, the pool descriptor is always available where it is needed.

Our pool allocator assumes that a memory pool consists of uniformly sized objects, but can allocate multiple consecutive objects if needed (for arrays). When pool allocating a complex data structure (for example, the main data structure for the `power` benchmark, shown in Figure 12(a)) each data structure node in the graph is allocated from a different pool in memory. This simple heuristic groups memory objects together of the same type, which works well for tree nodes, linked lists, and other heavily recursive structures. In the `power` benchmark example, there are four memory pools, each corresponding to a level of a heterogeneous "tree" structure, where each level is a linked list of nodes.

In addition to bookkeeping information for the pool allocator runtime, the pool descriptors can also be augmented to include pointers to the other pool descriptors in the data structure, form-




(a) Data structure graph (b) Pool descriptor graph

**Figure 12: Main data structure for `power` benchmark**

ing a graph isomorphic to the data structure graph (but accessable at runtime, and including backedges as well as forward edges). For the `power` benchmark, this graph is shown in Figure 12(b). By using this graph, the runtime can locate all of the memory blocks allocated to a data structure by traversing the pool descriptors for the data structure, inspecting the "`isAllocated`" bit for each cell that is allocatable from the pool. This information is very useful for a variety of transformations, for example the pointer compression algorithm described in Section 5.1.

## 3.2 Identifying candidate data structures

In order to pool allocate a data structure, we must detect the bounds on the lifetime of the data structure (to allocate and delete the pools themselves), and determine whether it is *safe* to pool-allocate the data structure. We use the data structure analysis graph for both purposes.

Using the data structure graph, we detect data structures whose lifetimes are bound by a function lifetime, allowing us to allocate the pool on entry to the function, and deallocate it on exit from the function. We identify these candidates by scanning the functions in the program (because each function's graph only contains the data structures that are acessable by that function), inspecting each functions interprocedural data structure graph as we go.

The lifetime of a data structure is contained the current function if the data structure's subgraph would be unreachable without the edges due to the scalar pointer map (i.e., no globals point to the structure, and it is not returned from the current function). This escape analysis (which is similar to the points-to escape analysis of [29]) is a conservative, but effective, heuristic for the appoximation of data structure lifetime. We refer to the function whose lifetime bounds the lifetime of the data structure as the "root" function, because it is the root of a subtree of the call graph that needs to be modified to handle pool allocation.

**PoolAllocateProgram**(program $Prog$)
  $\forall$ function $Fn \in Prog$
    $\forall$ disjointdatastructure $DS \in$ DSGraph($Fn$)
      if CallNodes($DS$) $\bigcup$ CastNodes($DS$) = $\emptyset$ then
        if $\neg$escapes($DS$) then
          PoolAllocate($Fn$, $DS$)

**Figure 13: Candidate identification algorithm**

It is safe to convert a data structure to use pool allocation when we can prove that we know all of the allocation and deletion points for nodes in the data structure, and that the program does not use the data structure in a non-typesafe way. To ensure that we have analyzed all of the relevant portions of the program, and that none of the accesses to the data structure contain unsafe operations, we do a simple traversal of the graph looking for either `call` or `cast` nodes. If either is found, a portion of the program is outside the scope of our analysis or an unsafe operation has been found, so pool allocation cannot be performed. This algorithm is shown in Figure 13.

## 3.3 Transforming function bodies

For a data structure identified to be pool allocated, the root function must be modified to allocate pool descriptors representing the various nodes in the subgraph. We insert code to stack-allocate a pool descriptor, code to initialize the pool descriptor on entry to the function, and code to destroy the pool descriptor (and all bookkeeping information associated with the memory pool) at the exit nodes of the function.

Once the pools are created, the body of the root function, and functions it calls (that use the data structure), must be transformed to use `poolalloc` and `poolfree` calls instead of `malloc` and `free` instructions. To do this the pool descriptor must be passed into called functions so that they are available for the eventual `poolalloc` and `poolfree` calls. The algorithm is shown in Figure 14.

**PoolAllocate**(function $RootFn$, datastructure $DS$)
  $Worklist = \{RootFn\}$
  $\forall$ function $Fn \in Worklist$
    $\forall$ instruction $I \in$ Instructions($Fn$)
      if UsesDataStructure($I$, $DS$) then
        if IsMallocOrFree($I$) then
          ConvertToPoolFunction($I$, $DS$)
        elseif IsCall($I$) then
          AddPoolArguments($I$, $DS$)
          $Worklist = Worklist \bigcup$ CalledFunction($I$)

**Figure 14: Function transformation algorithm**

The transformation loops over a worklist of functions to process, transforming each function until the worklist is empty. Initially the worklist is seeded with the root function, and is expanded whenever a call to an untransformed function is encountered. The body of a function is transformed according to the following rules:

- `malloc` and `free` operations referring to the pool allocated data structure are changed into calls to the `poolalloc` and `poolfree` library functions.
- Function calls that take a pointer into the data structure as an argument, or return a pointer that is part of the data structure are modified to pass the pool descriptor of the data structure into the called function. If the function has not already been processed, it is put on the transformation worklist.

The transformed `ProcessLists` function (Figure 15) allocates four memory pools, one for each data structure node in the two disjoint data structures in it. The `addList` function is transformed similarly.

One problem with modifying functions in this way is that it is possible for it to cause exponential code growth, because functions must be cloned before they are transformed. For example, if there is one client of `addList` that pool allocates the linked list, but another client that cannot, two copies of the code would be required. In practice, our implementation chooses to only pool-allocate a data

| Benchmark Name | LOC | Primary data structure | Analysis Time (millisec.) | Graph size for `main` | Primary DS size |
|---|---|---|---|---|---|
| `bisort` | 348 | binary tree | 47.3 | 21 | 1 |
| `em3d` | 683 | lists, arrays | 221.4 | 59 | 5 |
| `perimeter` | 484 | quad tree | 177.0 | 16 | 1 |
| `power` | 615 | hierarchy of lists | 59.2 | 21 | 4 |
| `treeadd` | 245 | binary tree | 13.5 | 19 | 1 |
| `tsp` | 578 | 2-d tree | 84.0 | 29 | 1 |
| `matrix` | 66 | 2-d matrices | 12.2 | 15 | 6 |

**Table 1: Preliminary Results for the Olden Benchmarks**

```
void ProcessLists(unsigned N) {
  PoolDescriptor L1PD, L2PD, P1PD, P2PD;
  List *L1, *L2;   unsigned i;
  poolinit(&L1PD, sizeof(List));
  poolinit(&L2PD, sizeof(List));
  poolinit(&P1PD, sizeof(Patient));
  poolinit(&P2PD, sizeof(Patient));

  L1 = (List*)poolalloc(&L1PD, 1);
  L2 = (List*)poolalloc(&L2PD, 1);

  /* populate lists */
  for (i = 0; i != N; ++i) {
    addList_pa(L1, &L1PD, poolalloc(&P1PD, 1));
    addList_pa(L2, &L2PD, poolalloc(&P2PD, 1));
  }
  useLists_pa(L1, &L1PD, L2, &L2PD);
  pooldestroy(&L1PD); pooldestroy(&L2PD);
  pooldestroy(&P1PD); pooldestroy(&P2PD);
}
```

**Figure 15: Source for pool-allocated** `ProcessLists`

structure if all the functions that refer to the data structure are only called with pool allocated data structures. Note that only functions that allocate or free nodes (or call functions that do) in the data structure need to be modified by this algorithm.

## 4. PRELIMINARY RESULTS

We have fully implemented both the disjoint logical data structure analysis and the automatic pool allocation transformation in the LLVM system. It performs fully automatic pool allocation for data structure graphs, including rewriting structure fields and scalar variables, pool allocation, and deallocation. It also supports allocation of a structure with heterogeneous nodes (using multiple pools).

We tested these transformations on several programs from the Olden benchmark suite, a collection of programs that have been used for several pointer prefetching studies (e.g., [19]). We present preliminary results for these programs here. Unfortunately, these codes are relatively small and each of them only allocates a single large, recursive data structure. Nevertheless, the benchmarks cover the full spectrum of difficult issues including pointer-intensive linked structures, dynamic stack allocation, pointers to functions, global variable references, and heavy use of recursion. In the future, we aim to test these techniques on other benchmarks, including the SPEC2000 benchmarks.

Table 1 shows the results for six of the Olden programs, and a simple matrix multiply routine function operating on three matrices. The table shows the compilation time for each benchmark, including the time for data structure analysis and pool allocation, but these times are nearly negligible because the benchmarks are quite small. For all the programs, the compiler is correctly able to identify the logical data structure used at the top level of the program. In all cases, however, lower-level functions in the tree allocate and process disjoint subsets of these logical structures, and the compiler correctly proves those subsets are disjoint (not reflected in the table). The interprocedural graphs computed for the top-level function (`main`) are relatively large but most of those nodes are due to calls to the external function `printf`, and the global format strings passed to those calls. The primary data structure in each benchmark had 4 nodes for `power`, 6 nodes for `matrix` and only 1 node for each of the other benchmarks. The structure in the `power` benchmark has 4 heterogeneous node types (as shown in Figure 4), and the compiler correctly identifies the linkages between these structures and allocates these to 4 separate pools, as noted previously.

In order to further demonstrate the capabilities of the compiler, we modified one of the benchmarks as follows. A potentially difficult usage pattern is that programs may allocate different structures and then repeatedly extract nodes from one structure and insert it into another. Nevertheless, such structures should still be reasonable candidates for separate pool allocation. To test the compiler in such a case, we wrote a simple code that creates two trees, initializes values in the nodes of both trees in one loop, and then traverses the two trees moving nodes from one to the other. The compiler again was correctly able to prove that the two instances are disjoint, and allocate the two trees to separate pools.

## 5. APPLICATIONS & FUTURE WORK

A primary benefit of automatic pool allocation is that it provides the compiler a basis for performing aggressive data structure transformations of entire linked data structures safely and transparently. Such transformations can use the data structure graph and knowledge about the pool allocation to analyze and optimize accesses to the data structures in the program at compile time, and to control the layout of data within pools at compile-time or runtime. We briefly discuss a few such transformations here, describing one of them (pointer compression) in some detail, but more work will be needed to implement and evaluate all these optimizations.

### 5.1 Pointer Compression

Only a small fraction of data structures in modern applications are likely to use $2^{32}$ or more addressable objects, and a significant fraction are likely to use less than $2^{16}$ objects. Pointer-intensive data structures on a 64-bit architecture therefore make very inefficient use of memory, and consequently memory bandwidth and cache capacity. For example, a tree structure with two child pointers and 8 bytes of data would require $3 \times 8$ bytes of memory. Replacing 64-bit pointers with 32-bit ones would reduce the memory consumption by $1/3$, and 16-bit pointers would reduce it by a factor of 2.

The logical data structure analysis and automatic pool allocation transformation make it possible to replace pointers with smaller index values into a pool. A simple transformation would be to use fixed 32-bit indices, and generate a runtime error if a single logical structure uses more than $2^{32} - 1$ separate objects at runtime. Although rare, such potential errors may make the transformation unacceptable to some large applications. A robust and more aggressive strategy is to dynamically grow the index sizes as needed, by relocating the pool and rewriting all indexes into it runtime. Such a strategy would allow us to use even smaller (16-bit) offsets initially, and grow these to 32-bit or 64-bit offsets if needed.

There are two major potential challenges to such a transformation. First, because the storage size of a pointer variable can change at runtime, some field offsets for structures containing pointers are no longer compile-time constants. This requires an additional level of indirection in addressing those structure fields, which can potentially cause high runtime overhead (this is the major challenge to achieving net benefits from pointer compression). The second challenge is that all object references must be dynamically rewritten when the offsets grow in size. Note, however, that this can happen at most 2 times for any single logical data structure ($16 \to 32$ and $32 \to 64$). We choose to limit ourselves to 16, 32, or 64-bit pointers for this reason. Here, we describe the basic transformation and discuss briefly how we can address these two challenges.

### *Pointer Size Transformation*

Any logical data structure for which automatic pool allocation is legal (as described previously) is a potential candidate for our pointer compression transformation. An additional restriction we impose is that pointers to objects of a type $T$ may be compressed only if *all* objects of type $T$ are pool-allocated, though not necessarily in the same pool. This ensures that the same code can be used to dereference all such pointers.

We assume for now that all objects in a single pool are homogeneous, i.e., different node types in heterogeneous structures are allocated in different pools. This may not be possible for object-oriented languages with inheritance, however, and we will have to make all pool entries be a multiple of the GCD of the static sizes of all types in the pool.

For a homogeneous logical structure, $S$, let its node type be $T$ and it's pool be $P$. The two previous analyses have already identified all static variables (scalar variables, structure fields, or array elements) that can hold pointers either to the nodes of that structure, or to fields within a node. If there is a such a variable within the type $T$ (i.e., type $T$ is recursive), or within any other pool that has a pool pointer to $P$, we simply change the type of that variable to hold an index into pool $P$. At compile-time, we use integers of size $O_{min}$ for the indices, where $O_{min}$ defaults to 16 bits.

If a pointer variable outside any pool is of type $T*$, it must be replaced with a pair of variables $<$*ptr(Pdesc), index*$>$, so that the pointer can "carry with it" the information required to decode the indices. The same transformation also applies to all formal parameters of type $T*$, so that pool descriptor pointers are passed along with indices on function calls (and returns).

If the index size is never changed at runtime, then the above simple transformations are sufficient. To enable dynamic size changes, more complex runtime support and address arithmetic are needed. The pool descriptor is augmented to include an array of offsets for each field of $T$. An access to $X \to \text{fld}_n$ ($0 \leq n <$ number of fields of $T$) becomes:

```
*(Pdesc→poolbase + Xindex + Pdesc→offsets[n])
```

Clearly, the overheads of such an access sequence could overwhelm any benefits of the transformation. The next section describes some optimizations that are crucial to make this strategy practical. The section following it describes how references are updated on a runtime pointer size change.

### *Optimizing Pointer Dereferences*

We propose two optimizations that can be performed at compile-time to achieve efficient structure field addressing in the presence of dynamically changing field sizes.

First, since accesses to type $T$ are type-safe, we can safely reorder the fields of $T$. We can move all non-pointer fields of $T$ to come before pointer fields. This guarantees that the offsets for non-pointer fields and for the first pointer field are compile-time constants, and involve no additional overhead beyond that of ordinary pool accesses, i.e., they only require the first two terms in the address expression above. The result is that many common-case accesses (e.g., non-pointer fields in flat structures, and the first pointer in any structure) have zero overhead for dynamic pointer compression.

The second key optimization is to move field access calculations out of loops or regions of the call graph that do not allocate or deallocate structure nodes. The pool allocation transformation already identifies the program points where these occur.

### *Dynamic Pointer Size Changes*

When a pool $P$ detects that it must be expanded and it's index range grows beyond that addressable by the current index size, we must rewrite all locations that contain indices into pool $P$. References within objects in $P$ can be found simply by traversing all the objects in $P$, all of which are of known types. The same operation suffices on every pool that contains references into $P$; such pools are easy to find because $P$ has pointers to all other pools containing references into it. Finally, the pool descriptors must be updated to record the new offsets for fields of types that contain pointers into this pool.

## 5.2 Improved Prefetching Strategies

One of the major challenges in optimizing accesses to linked data structures is to effectively hide memory latencies via prefetching. One key difficulty is to predict which objects will be referenced in the future without explicitly loading a chain of pointer values; this has been called the "pointer-chasing problem" [19]. A variety of solutions have been proposed, most of which add a "history pointer" to each structure node to record either the allocation order or some expected traversal order for linked data structures. These history pointers are then used to schedule prefetches for objects expected to be accessed later in the execution [19, 4, 21]. The automatic pool allocation and pointer compression transformations each enable a potentially valuable improvement for such prefetching strategies.

**Creation-order prefetching without history pointers**: One of the key history-pointer prefetching schemes previously proposed is creation-order prefetching, in which history pointers are used to record the order in which nodes in a linked structure are created at runtime and to prefetch nodes in that order. Luk and Mowry suggested a variant of this scheme they call Data-Linearization Prefetching, which would not require history pointers, and instead relies on placing consecutively created objects within the data structure in consecutively memory focations (or remapping them after allocation), but they do not describe how to do so in a compiler [19].

In fact, the automatic pool transformation is exactly what is needed to implement such a scheme, i.e., to perform creation order prefetching *without adding any history pointers to the data structures*. In particular, if memory for successively allocated objects in a pool is allocated in consecutive memory locations, later traversals of the linked data structure can prefetch values $k$ accesses ahead for arbitrary $k$, without needing any history pointers. This would not work for structures whose linkages are modified after the initial allocation (which the compiler could easily detect from the previous analyses), but could work well for structures that are created and then repeatedly traversed in order.

**History pointer prefetching with near-zero memory overhead**: When the traversal order for a structure may not match the creation order, history pointers of some kind must be used in order to capture information about the expected traversal order. The pointer size transformation makes it possible to use history-pointer prefetching *with nearly zero memory overhead* relative to the original application: we simply use the space saved in a data structure for recording history pointers.

The software-based history pointer schemes require the compiler to add extra pointer fields within a pointer-based data structure, and these fields hold pointers to other elements of the *same* data structure. Therefore, these extra pointers can also be stored in current compressed format at any point in time. Furthermore, all schemes we know of require at most one extra pointer for each pointer in the original structure. Our pointer compression schemes guarantee at least a factor of two reduction in memory usage per pointer. The previous two facts imply that we can add history pointers to a compressed structure completely free of any memory overhead (except the negligible space used by the pool descriptors). We are not aware of *any existing software schemes* that can perform history pointer prefetching without signficant memory overhead.

## 5.3 Automatic Small-scale Parallelization

Small-scale parallelism is increasingly important because of several key architectural trends, including widespread use of multiprocessor servers with 2 to 32 processors, the emergence of processors based on simultaneous multithreading [28] or on-chip multiprocessing, and the fine-grain explicitly parallel architecture (EPIC) used in the Itanium Processor Family. The techniques proposed in this paper potentially enable us to extract small degrees of relatively coarse-grain parallelism for ordinary programs. In particular, if the compiler proves that two logical data structures are disjoint and have no cross references, it follows that updates of those two structures can be performed in parallel (if computations on one structure do not use values from the other). This is a different style of parallelism than that extracted by other parallelization strategies for non-array-based programs [20] and for tree-like data structures [17]. Those strategies focus on data parallelism within computations on the same data structure or set of structures, whereas we would obtain parallelism between computations on different structures. This might only yield small degrees of parallelism, but such parallelism may in fact be well-suited to the emerging systems described above.

# 6. RELATED WORK

There is a rich literature on pointer analysis techniques and on shape analysis of data structures in programs. Our disjoint data structure analysis is most similar to previous work on computing heap approximations such as path matrices [17] and static shape graphs for heap storage (e.g., [14, 18, 22]). One key difference between all these previous analyses and ours is that the previous approaches are flow-sensitive — they compute a heap approximation (e.g., one or more shape graphs) for each program point. In contrast, we only compute a single summary graph for an entire procedure. This also means that our analysis does not have to be iterative. We also use a significantly simpler interprocedural strategy than previous papers, which generally either perform dataflow analysis on a full interprocedural control flow graph (e.g., [18]) or an iterative analysis on the call graph (e.g., [17, 14]). In contrast, we never build either a call graph or an interprocedural control flow graph. Instead, we handle both direct and indirect function calls simply as additional memory objects in our summary graphs, and inline the summary graph for a callee at each call site to achieve a simple but powerful context-sensitive analysis that is linear in the number of call sites and call edges in the program.[3]

Many context-sensitive pointer analysis techniques also compute a path-based or graph-based approximation to the pointer reachability properties of pointer variables and heap objects (e.g., [11, 5, 29]). These techniques generally do not identify logically disjoint data structures, even though they can probably be extended to do so. For example, some techniques use a single name to represent all heap objects at a particular allocation point [11, 29]. Others only compute alias relationships between different pointer variables using a "store-less" approach, and do not attempt to derive a model of heap storage [5]. In both cases, the algorithms could probably be extended in a fairly straightforward manner to identify disjoint heap-based data structures.

We believe that the automatic pool allocation transformation proposed here is the first such algorithm for programs containing explicit uses of malloc and free to manage heap memory. As noted, pool-based allocation is a widely applied manual technique. Language support has been developed for simplifying the use of pool-based allocation via program annotations, and for checking the correctness of pool usage through enriched type systems [10, 13]. There has been previous work on automatically identifying regions of memory for ML, i.e., for a type-safe functional language without side-effects. Their analysis is used as either the sole [26, 25] or the primary [16] means of releasing dynamically allocated memory back to the system. They also simplify the problem by using execution scopes to identify regions as collections of objects that become unreachable at the same scope, and do not consider arbitrary region lifetimes.

In contrast, our primary goal is to use automatic pool allocation as a means of enabling further compiler optimizations for linked data structures. Consequently, a key difference in our work is that we automatically introduce pool-based allocation for a wider range of (but not necessarily for all) heap-based data structures, including long-lived structures with arbitrary lifetimes (but with type-safe usage). Furthermore, we correctly handle programs that rely on explicit malloc and free operations. Although we could use our analysis information to eliminate explicit deallocations of data structure nodes, we choose not to do so for now. Instead, we allow the program to free objects in the pool, but also release all memory for a pool back to the system when when the data structure is unreachable.

There is a broad class of conservative program transformations that attempt to give better locality to programs by changing the placement of allocations in memory [9, 15, 2, 7, 23]. In contrast to these techniques, our approach uses an *exact* assignment of allocated objects to pools. While approximate assignment algorithms do provide the locality properties we are seeking, the lack of exact assignment makes them unsuitable to host more aggressive transformations, such as pointer compression.

Finally, to our knowledge, there is no previously published work on safe, automatic compression of pointer variables. This is a novel optimization enabled by automatic pool-based allocation. The most similar system is described in [32], which uses programmer annotations to identify integers that are usually small and pointers that

[3] In fact, we believe that our analysis is linear in the size of the program under reasonable assumptions about structure fields and nested structure types, but proving that requires further work.

usually contain a small offset from the current object. Unlike our work, that paper requires the program to provide the safety analysis, and hardware support to efficiently access and detect cases where the underlying values exceed their allocated ranges.

# 7. CONCLUSIONS & FUTURE WORK

This paper presented a novel transformation that converts ordinary programs to use pool-based memory allocation. The transformation relies on a link-time interprocedural analysis to identify disjoint logical data structures in the program, and to check whether these data structures are accessed in a type-safe manner. Even though this is a context-sensitive interprocedural analysis, it is simple and quite efficient because only small summary graphs are propagated between procedures, interprocedural information is only propagated from callees to callers, function pointers are handled uniformly with other memory pointers during the analysis, and the analysis is flow-insensitive.

The disjoint data structure analysis combined with automatic pool allocation can enable some sophisticated macroscopic optimizations for linked data structures. We described a novel automatic pointer compression transformation and briefly discussed several other optimization possibilities enabled by this work. A key goal of our ongoing work is to implement and evaluate these optimizations.

# 8. REFERENCES


[1] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, 33(5):301–312, 1998.

[2] D. A. Barret and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proc. SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 187–196, Albuquerque, New Mexixo, June 1993.

[3] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):367–399, 1993.

[4] B. Cahoon and K. McKinley. Data flow analysis for software prefetching linked data structures in java. In *International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, Sept. 2001.

[5] B.-C. Cheng and W. mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evalutation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, Vancouver, British Columbia, Canada, June 2000.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[7] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. *ACM SIGPLAN Notices*, 34(3):37–48, 1999.

[8] I. Corporation. Xl fortran: Eight ways to boost performance. White Paper, 2000.

[9] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, 1988.

[10] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.

[11] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994.

[12] M. F. Fernández. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Notices*, 30(6):103–115, 1995.

[13] D. Gay and A. Aiken. Language support for regions. In *Proc. SIGPLAN '01 Conf. on Programming Language Design and Implementation*, pages 70–80, Snowbird, UT, June 2001.

[14] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[15] D. Grunwald and B. G. Zorn. Customalloc: Efficient synthesized memory allocators. *Software - Practice and Experience*, 23(8):851–869, 1993.

[16] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[17] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed System*, pages 35–47, 1990.

[18] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 21–34, July 1988.

[19] C. Luk and T. Mowry. Compiler-based Prefetching for Recursive Data Structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Boston, USA, Oct. 1996.

[20] M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, 1997.

[21] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.

[22] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1), Jan. 1998.

[23] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 12–23. ACM Press, 1998.

[24] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[25] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(1), 1998.

[26] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, pages 132(2):109–176, Feb. 1997.

[27] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 322–329, Oct. 1998.

[28] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[29] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Snowbird, UT, June 2001.

[30] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proceedings of CC 2000: 9th Int. Conf. on Compiler Construction*, Berlin, Ger., Mar-Apr 2000.

[31] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the conference on Programming language design and implementation*, pages 177–191. ACM Press, 1991.

[32] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *International Conference on Compiler Construction (CC)*, Apr 2002.